CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- HW9 posted, and will be due on next Tuesday (Nov 12th) midnight (11.59pm)
- Will have you take-home Quiz 3
  - Post by tomorrow, and
  - due on Sunday 11.59pm
  - Submit onto Blackboard

Recap

- Binary Heap:
- Max_heap vs. Min_Heap
- Reheapify:
  - Trickle up
  - Trickle down
- Two basic operations:
  - Insert: possibly need to be trickling up
  - Delete: possibly need to be trickling down

Learning Topics

- Heap Sort
- Hash Table

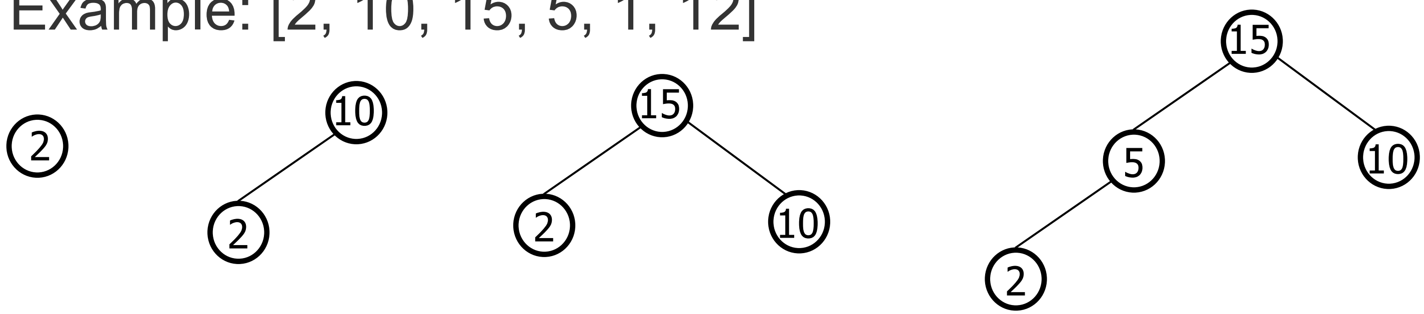
Heap Sort Idea

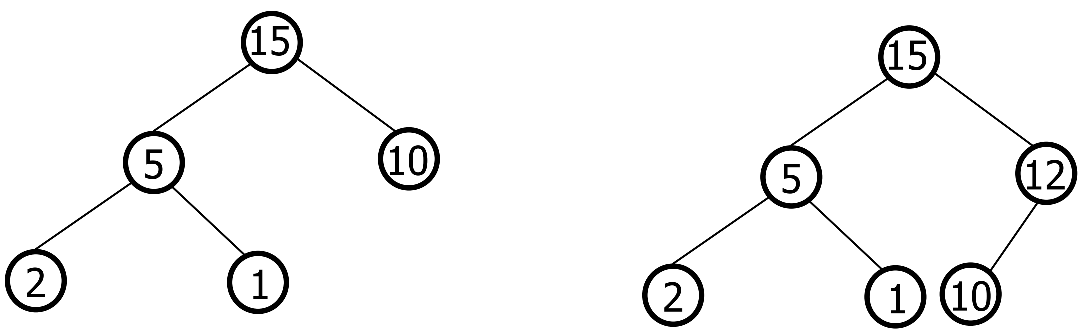
- How to use heap to sort an array?
  1) Make a heap out of array
  2) Remove (successively) from the heap

- Given an array, how to heapify it?
- Two possible ways to do that:
  - Successive insertion
  - Start with random placement, rearrange them to satisfy heap ordering

Making a Heap Using trickleUp()

- For each item, i, in array call insert(i)
- Example: [2, 10, 15, 5, 1, 12]



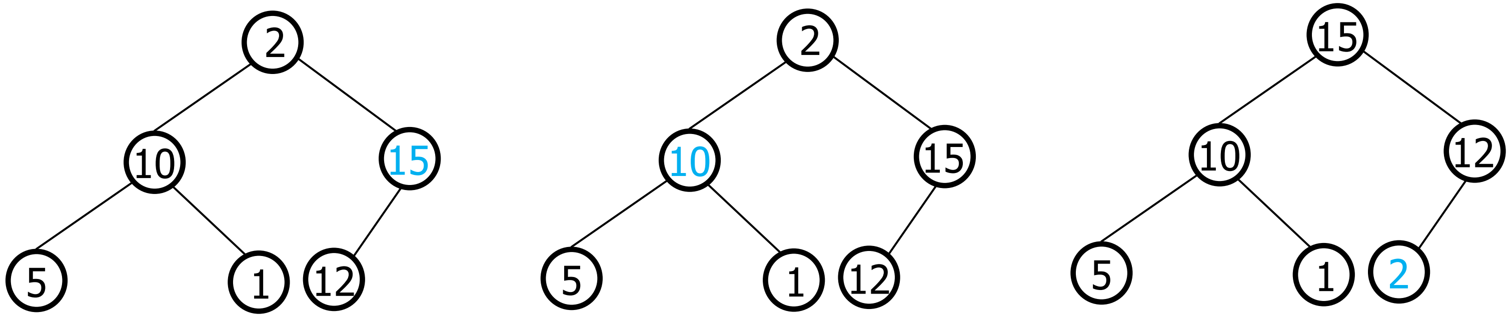


Output: [15, 5, 12, 2, 1, 10]

What is the time complexity? O(n logn) we insert/trickleUp n times, each is O(logn)

Making a Heap Using trickleDown()

- Start with a random placement, then rearrange using trickleDown()
- Example: [2, 10, 15, 5, 1, 12]



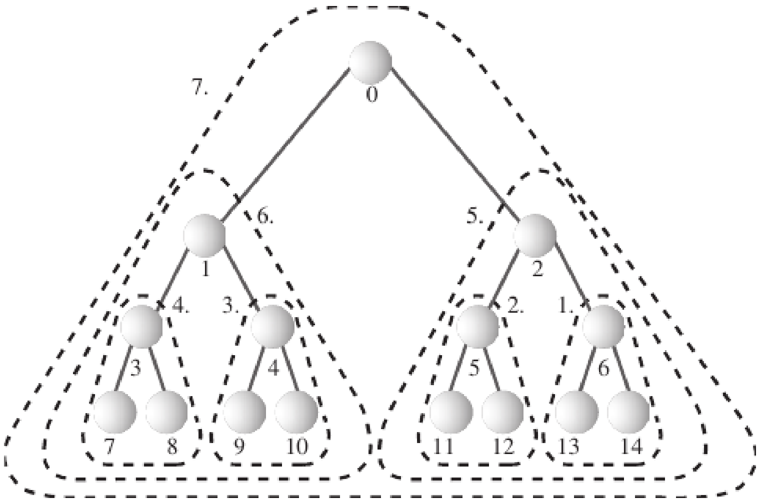
Output: [15, 10, 12, 5, 1, 2]

Notice that we got a different heap compared to using trickleUp()

Making a Heap Using trickleDown()

- How many times do we call trickleDown()?
  - n/2 times, no need to trickleDown leaves

```
for (int i=size/2-1; i>=0; i--)
   trickleDown(i);
```



- What is the time complexity?
  - O(n logn)
  - But, the tight bound is O(n).
  - To see why, think about why it is better to trickleDown() compared to trickleUp().

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 603

Heap Sort

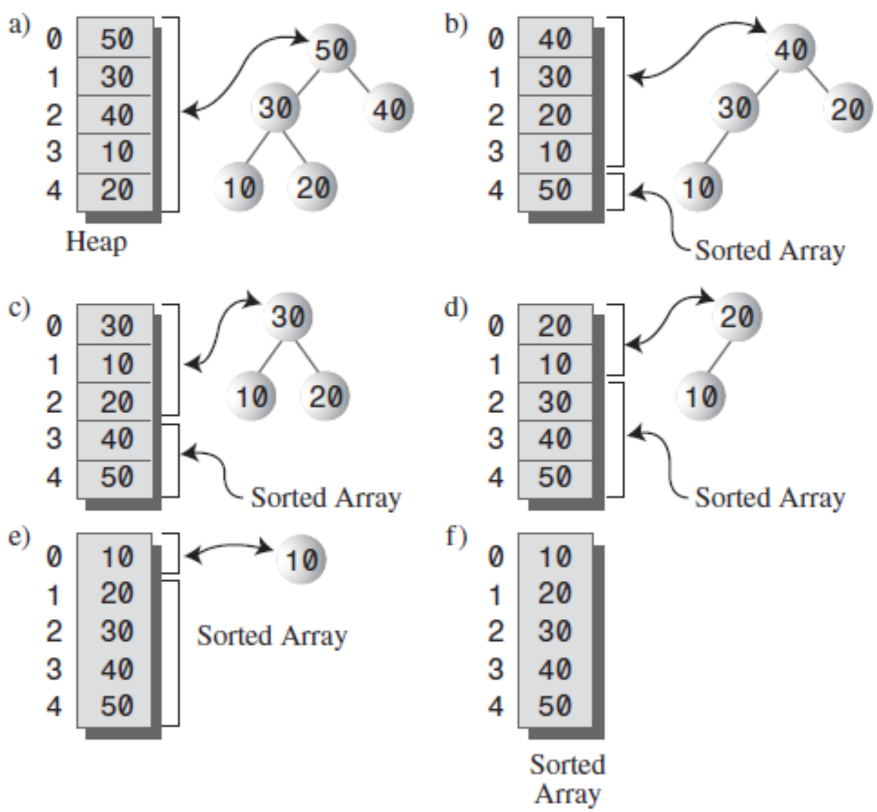
1) Make a heap out of the input array
   - Can be done in O(n) using trickleDown()
   - It is in-place
2) Successive removal from the heap
   - Removing one element takes O(log n)
   - We remove n elements. So, the total complexity is O(n logn)

- The entire process can be in-place

```
for (int j=size-1; j>=0; j--)
{
   Node biggestNode = remove();
   heapArray[j] = biggestNode ;
}
```

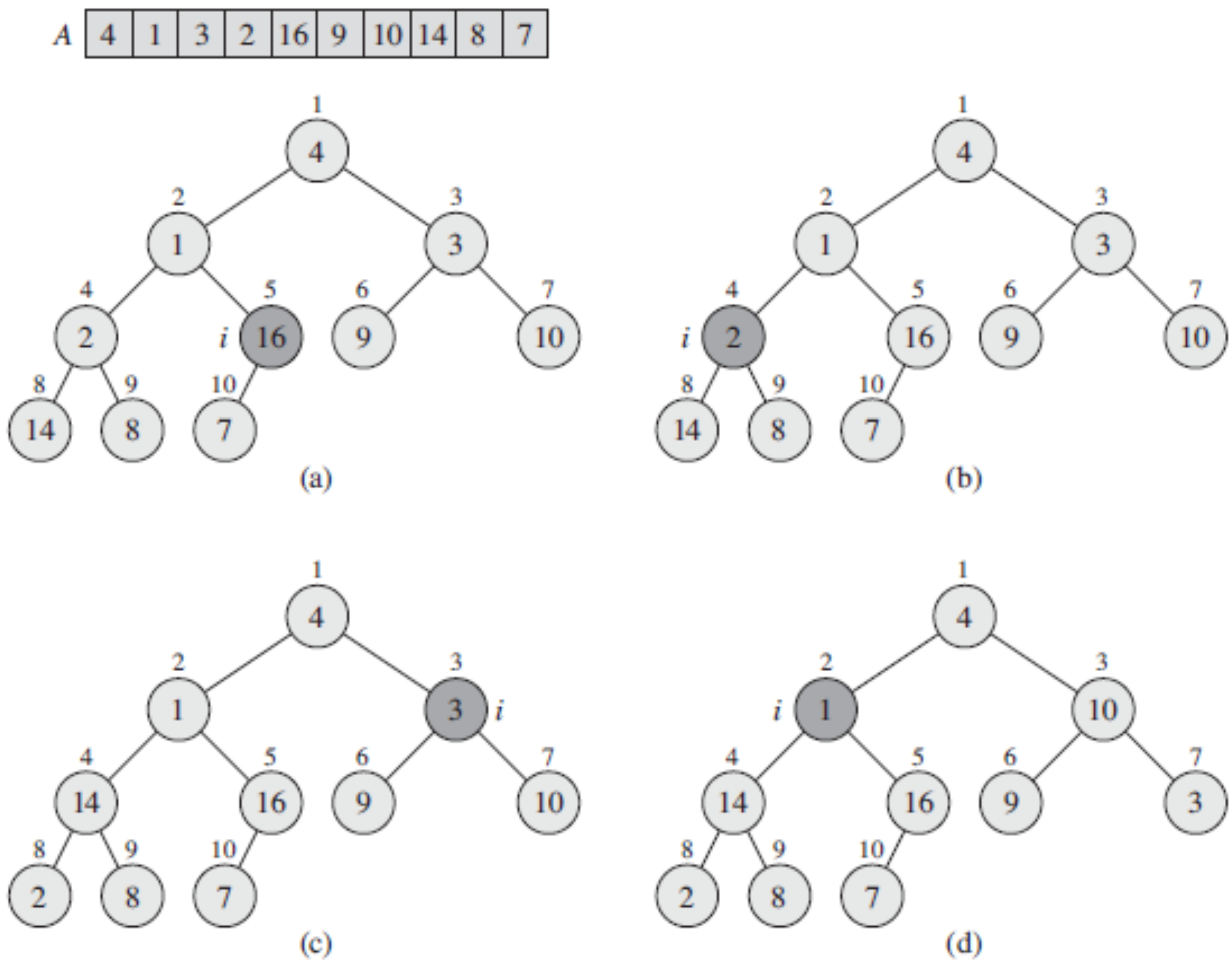
In-place Heap Sort

- Put the removed element at the end.

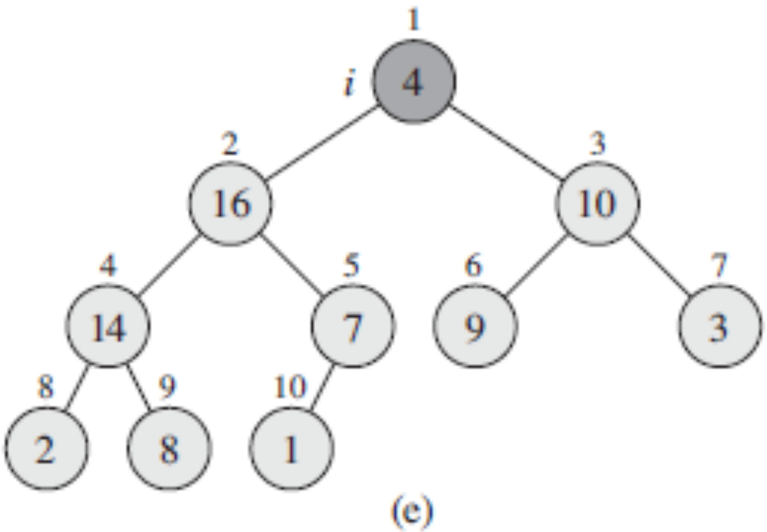


Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 604

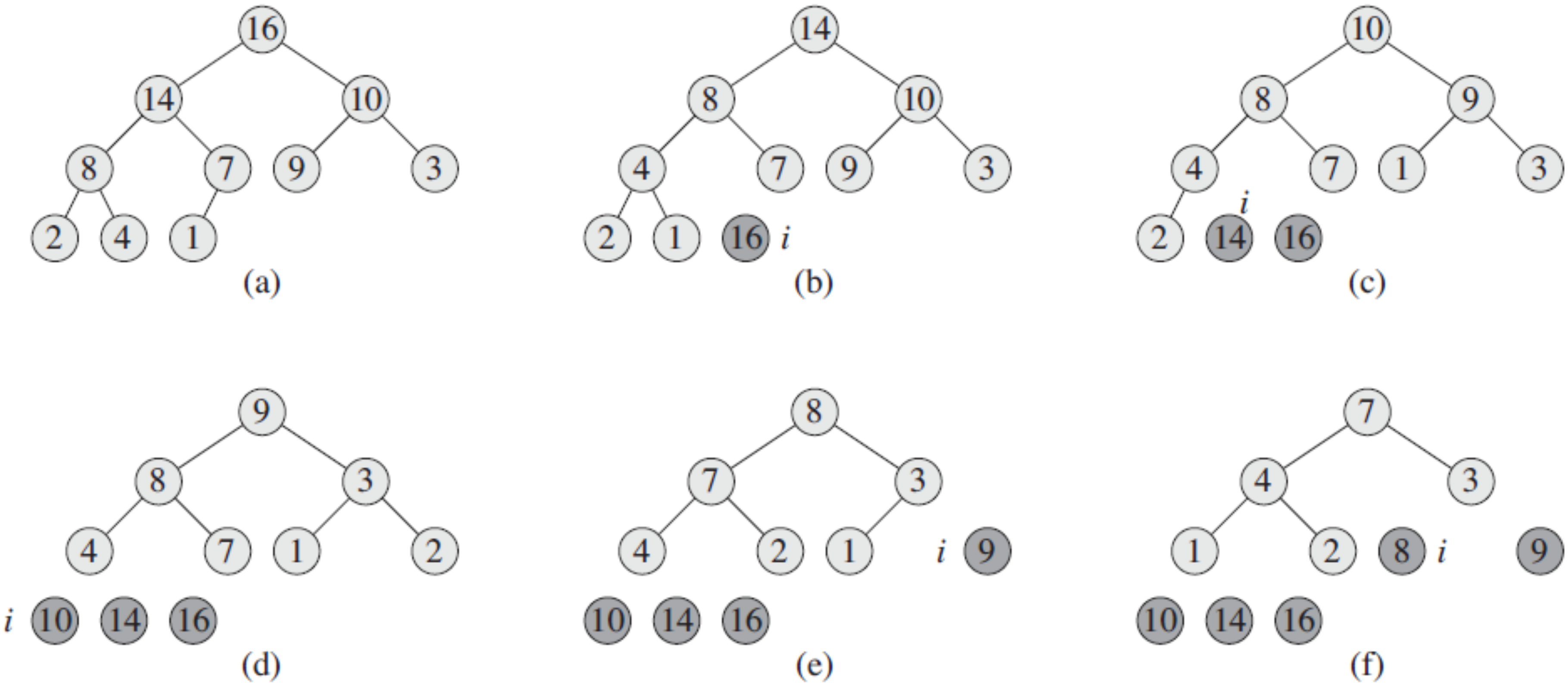
Another Example:



Another Example:



Another Example:



Algorithm Analysis

<u>*HeapSort*(*A*)</u>

1. Build-Max-Heap(*A*)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. *MaxHeapify*(*A*, 1)

- In-place
- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Sink takes time $O(\lg n)$.
- Therefore, $T(n) = O(n)+O(n \lg n)=O(n\log n)$
- For best, worst and average case.

Heap Sort: Performance Analysis

- The height of the heap: (int)$\log_2 N$
- No. of nodes of

  height h $\leq \lceil n/2^{h+1} \rceil$
- Heap sort:
  - Build-Max-Heap(A)
  - N-1 times:
    - Sink(A, i, N)

Running Time of *Build-Max-Heap*

- Loose upper bound:
  - Cost of a *Sink call* × No. of calls to *Sink*
  - $O(\lg n) \times O(n) = O(n \lg n)$
- Tighter bound:
  - Cost of a call to *Sink()* at a node depends on the height, *h*, of the node – $O(h)$.
  - Height of most nodes smaller than *n*.
  - Height of nodes *h* ranges from 0 to $\lfloor \lg n \rfloor$.
  - No. of nodes of height *h* is $\lceil n/2^{h+1} \rceil$

Running Time of *Build-Max-Heap*

Tighter Bound for *T*(*BuildMaxHeap*)

T(*BuildMaxHeap*)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$
$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$
$$\leq \sum_{h=0}^{\infty} \frac{h}{2^h} \qquad , x = 1/2 \text{ in (A.8)}$$
$$= \frac{1/2}{(1-1/2)^2}$$
$$= 2$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$
$$= O(n)$$

Can build a heap from an unordered array in linear time

Heap Sort

- Good property:
  - Running time: ~O(nlogn)
  - Memory usage: in-pace. No extra space needed.

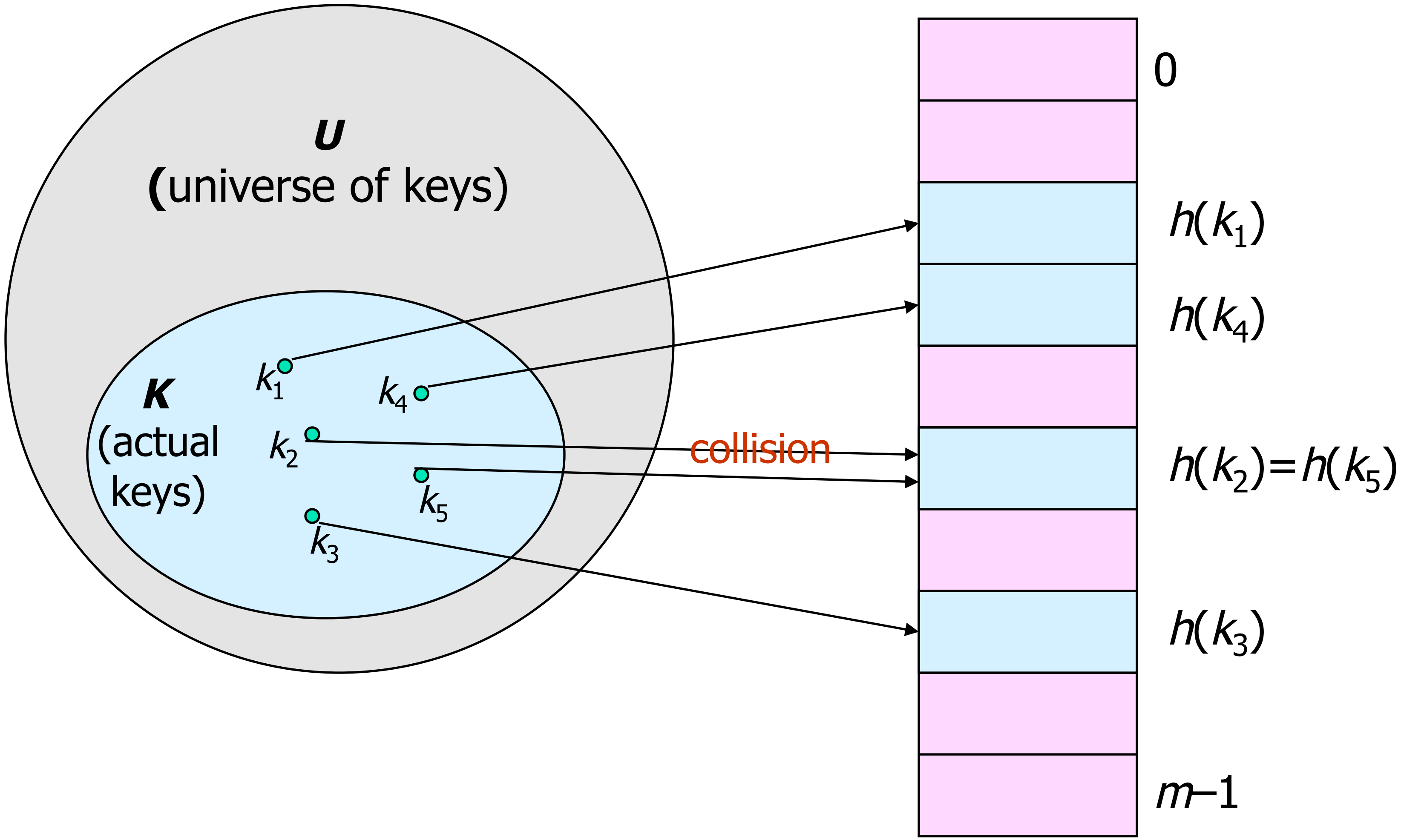
Hash Table

- A very useful data structure, great for efficient lookup of values given a key.
- What data structure to use for fast insertion and searching?
  - Unordered array
    - Insert $O(1)$, search $O(n)$
  - Ordered array
    - Insert $O(n)$, search $O(\log n)$
  - Linked list
    - Insert $O(1)$, search $O(n)$
  - Balanced BST
    - Insert $O(\log n)$, search $O(\log n)$
  - Hash Table
    - Insert $O(1)$, search $O(1)$

Hash Table and Hash Functions

- Hash Table:
  - Array (called “table”) of size ***M***, to hold keys (or key-value pairs);
  - Need a hash function ***h(k)*** to transform a key ***k*** into an index in the array: an integer between [0,M-1]
    - such integer h(k) is the hash value of k.
  - We can store key (k) or key-value pair (***k***, ***v***) at index ***i*** = ***h***(***k***)
  - We can use the Hash Table to store a large range of keys into a table of smaller range.

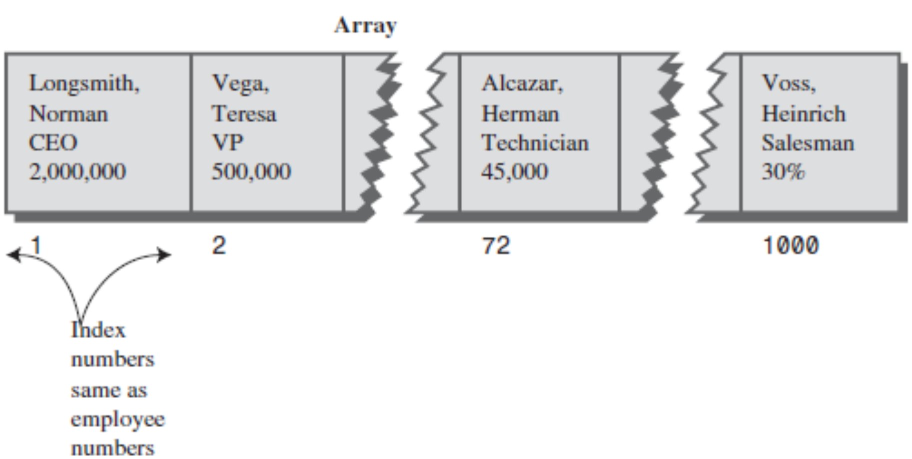
Hashing



U
(universe of keys)
K
(actual keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

Hash Table- Example Scenario

- Employee records
- Use an array to store employee objects, employee ID numbers (*keys*) can be chosen as the index numbers.
- O(1) for insertion and search



What if the range of employee IDs are wide and not all of them are valid?
It will result in a very inefficient use of space.

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 520

Hash Table Example

- We store employee records at index obtained by computing hash(Employee ID) which is Employee ID%100.
- We can access an employee record by going directly to the corresponding index (hash(Employee ID))

	Employee ID	Name	Title
0			
:	:	:	:
25	52784025	Jake	Data Analyst
:	:	:	:
64	67839164	Sara	CTO
:	:	:	:
99			

Hash Function

- What is a good hash function:
  - Consistent: equal keys will produce the same hash value;
  - To be deterministic
  - Efficient to compute
  - Uniformly distribute the keys, so that collision can largely be avoided.

- Modular hashing: using the modulo operator (%)

$$h(k) = k \bmod M$$

Hash function: Modular Hashing

- Modular hashing

  $$h(k) = k \bmod M$$

  - How to choose a good key
    - Example1: phone number
      - Bad: the first three digits; Better: the last three digits;
    - Example2: Social Security Number
      - Bad: first three digits; Better: last four digits;
  - How to well-represent the key, especially how to deal with non-integer keys.
    - Like Strings
  - How to choose M

Modular Hashing: choosing *M*

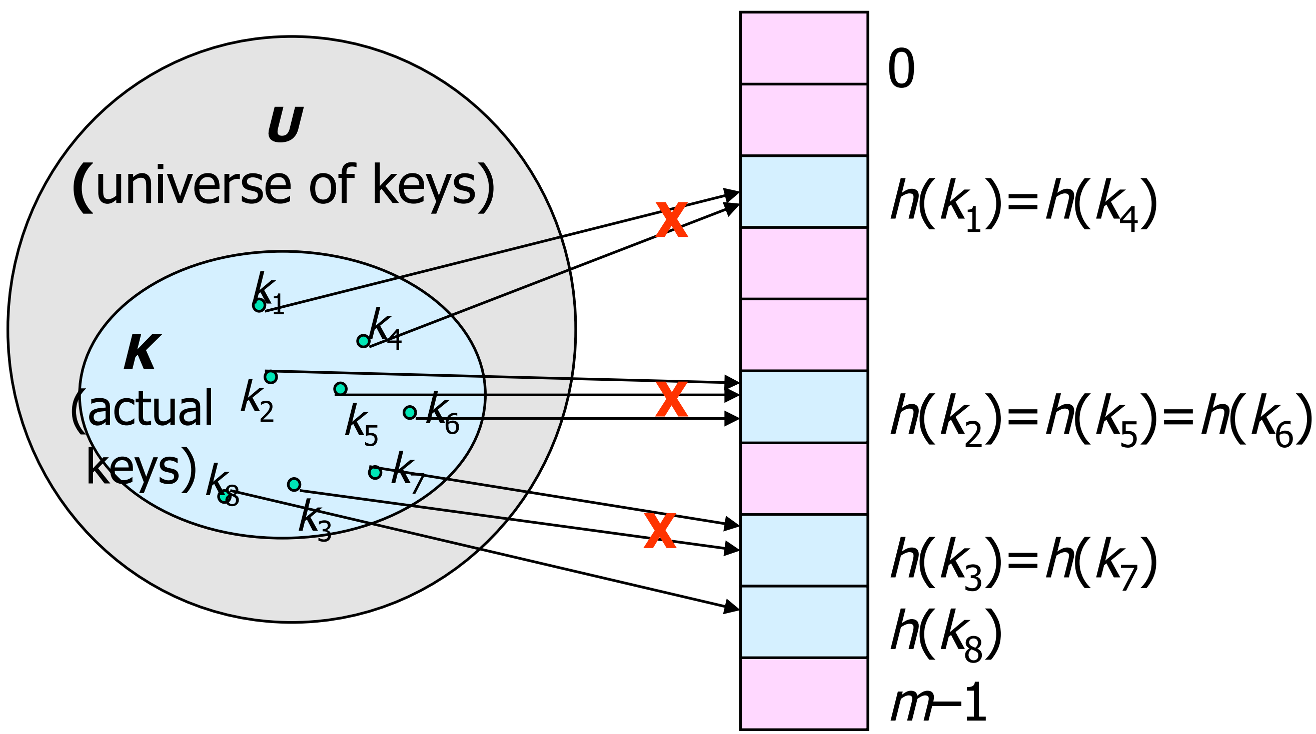
Advantage:
•Fast, since requires just one division operation.
Disadvantage:
•Have to avoid certain values of *m*.
•Don't pick certain values, such as $m=2^p$ or $m=10^p$
•Or hash won't depend on all bits of *k*.
Good choice for *m*:
•Primes, not too close to power of 2 (or 10) are good.

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Collision Resolution

Collision happens when distinct keys hash to the same array index.

- Collision cannot be completely avoided and they are evenly distributed.
- e.g. people's birthdays.

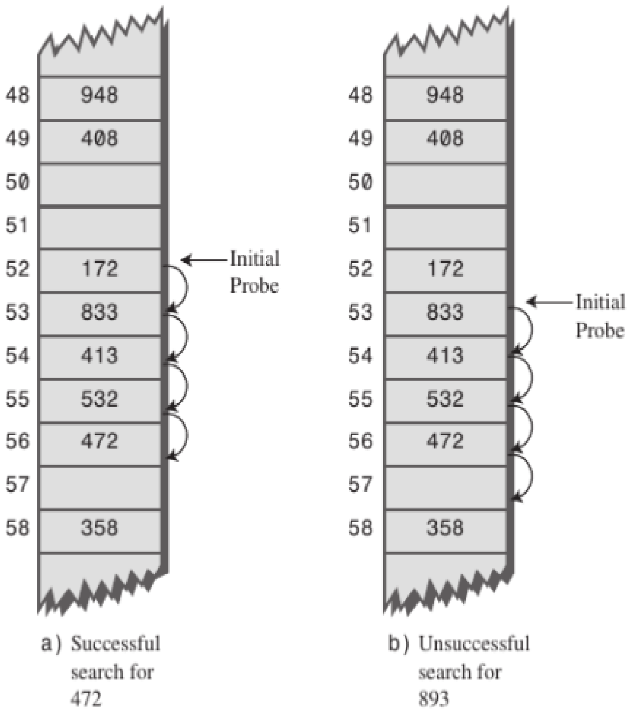


Hash Table - Collision

- How to address collisions?
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Separate Chaining

Linear Probing

- Insert: compute the hash, if collision, keep incrementing the probe until an empty spot is found in the array.
- Search: compute the hash, increment the probe until you find the key or you see an empty spot(search failure)
- Delete: find the key, mark it as deleted, for instance use a special key -1
  - Insertion considers the deleted array cell as empty.
  - Search considers the deleted array cell as occupied.



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 531

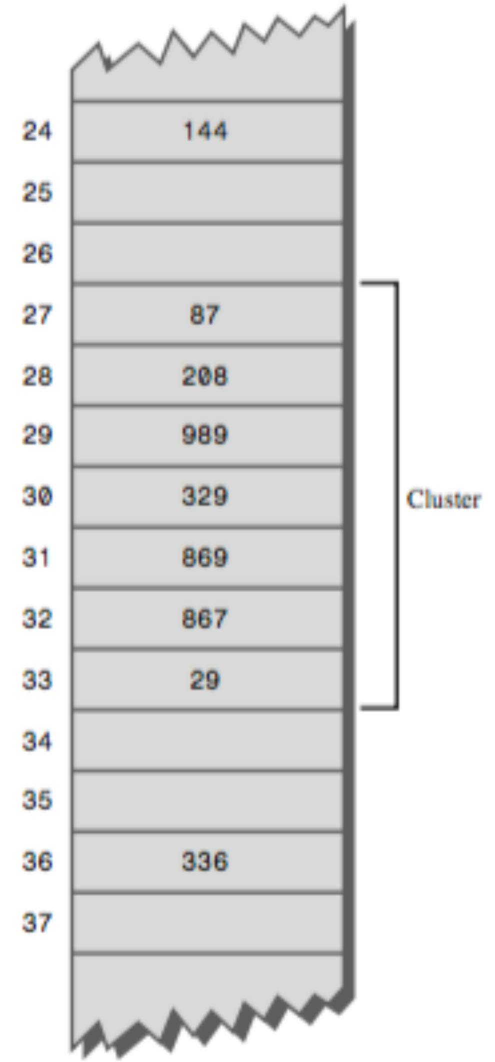
Linear Probing

```
class HashTable{
   private DataItem table;
   private int size;
   public HashTable(int size){
      this.size=size;
      table = new DataItem[size];
   }
   public int hash(int key){return key%size; }

   public void insert(DataItem item){
      int h = hash(item.getKey()); // key
      while (table[h]!=null && table[h].getKey()!=-1) { // until empty spot or -1
          h=(h+1)%size;//if occupied,increment by 1
      }
      table[h]=item; // enter item into table
   }

   public DataItem find(int key){
      int h = hash(key); // compute hash of item
      while (table[h]!=null && table[h].getKey()!=key) { // find matching item or null
          h=(h+1)%size; // if no match, increment by 1
      }
      return table[h]; // return item or null
   }
}
```

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 535

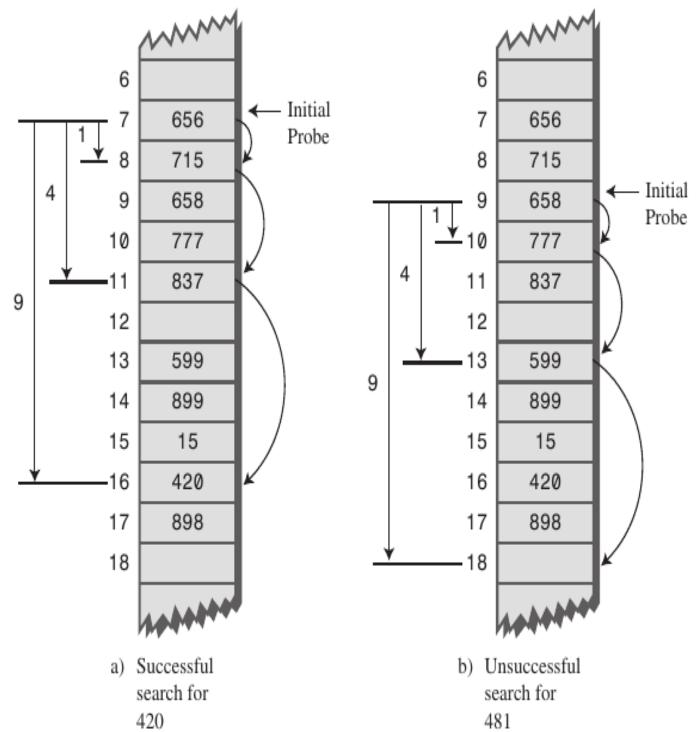
Linear Probing- Comments

- Issue of *clustering*: after collision, two adjacent array spots a
occupied, it is very likely that this forms a growing cluster.
- A cluster is a sequence of filled cells on an array.
    - Results in very long probe length and degrades perform
- Load factor: the number of items in hash table(n) divided by
the table(b); n/b.
    - When the load factor increases performance of linear probing drop
    - There is a trade off between space efficiency and hash performanc
    - Keep load factor between 1/2 to 2/3
- Resizing the array
    - We cannot easily copy items into a bigger array. Need t
hashing, which takes O(n)



Quadratic Probing

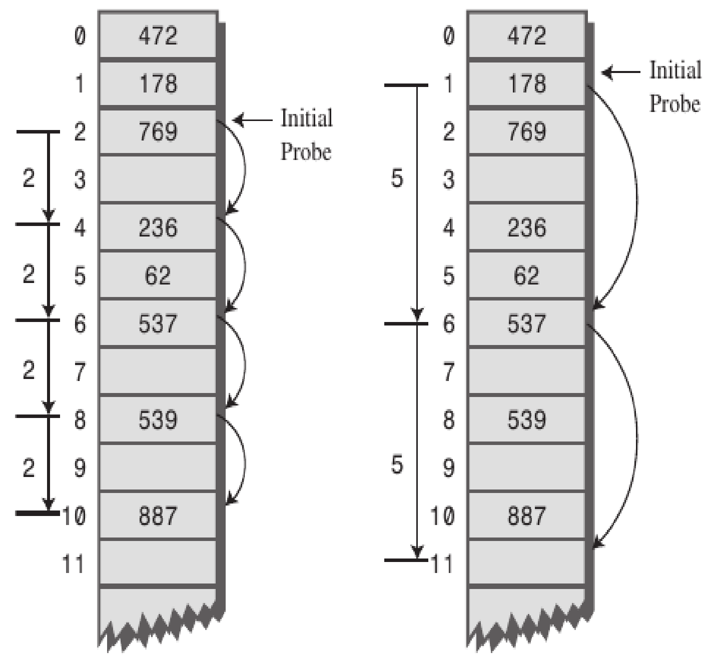
- Quadratic probing
  - In linear probing: hash, hash+1, hash+2, hash+3,…
  - In quadratic probing: hash, hash +1^2, hash+2^2, hash+3^2,…
- It tries to avoid cluster formation by jumping over adjacent array cells.
- Performance degrades at higher load factors compared to linear probing
- Clusters can still be formed, keys follow the same jumping pattern.



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 543

Double Hashing

- The main idea is to relate the probe sequence to the data/key, instead of making it fixed for everyone.
- It uses two different hash functions.
  - First one determines the initial index
  - Second hash determines the step for probing
    - It cannot output 0 as step size
- Use a prime number as the array size
  - Example of non-prime array size 15
    - initial index of 0, step size of 5. The probe sequence will be 0, 5, 10, 0, 5,10, and so on.



a) Successful search for 887

b) Unsuccessful search for 709

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 545

Double Hashing

```
public int hash1(int key){
  return key%size;
}
public int hash2(int key){
  return 5-key%5;
}

public void insert(DataItem item){
   int h = hash1(item.getKey()); // key
   int s = hash2(item.getKey()); // step
   while (table[h]!=null && table[h].getKey()!=-1) {//until empty spot or -1
       h=(h+s)%size;//if occupied,increment by s
   }
   table[h]=item; // enter item into table
}

public DataItem find(int key){
   int h = hash1(key); // compute hash of item
   int s = hash2(key); // compute step
   while (table[h]!=null && table[h].getKey()!=key){//find matching item or null
       h=(h+s)%size; // if no match, increment by s
   }
   return table[h]; // return item or null
}
```

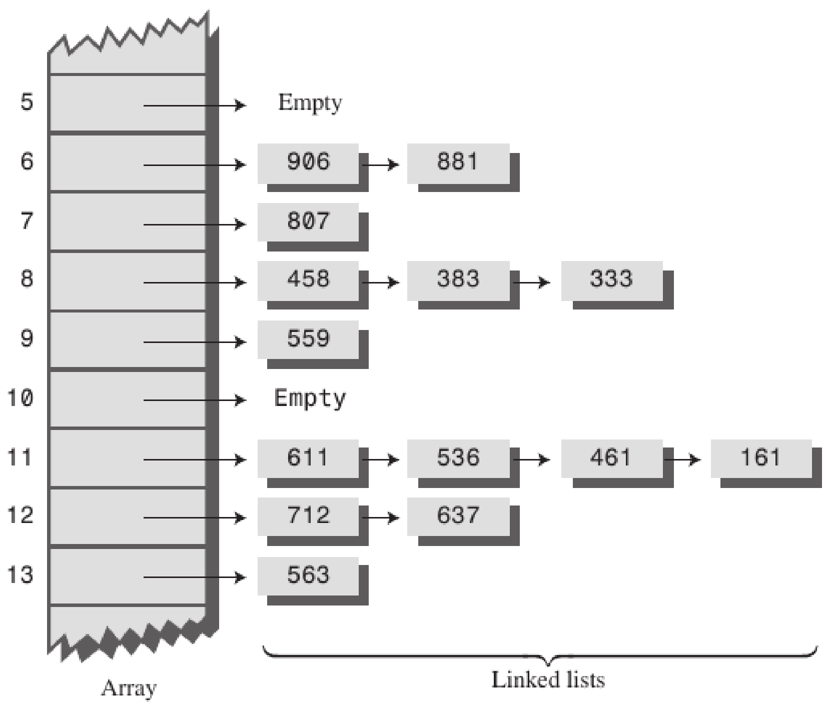
Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 546

Efficiency analysis: Open addressing liner probing

- Performance: depends on the constant factor α=N/M
- The average number of probes for search hit is: $$\frac{1}{2}(1+\frac{1}{1-\alpha})$$
- The average number of probes for search miss is: $$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$$
- We normally set M=2N

Separate Chaining

- The idea is to have a linked list at each index
- When collision occurs, add to the linked list



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 552

Separate Chaining- Comments

- The idea is simple, but more involved implementation.
- If the list is too long, performance is not good.
- It can tolerate higher load factors compared to open addressing.
- It takes more memory compared to open addressing.

Comments on Hash Tables

- It is all about a good hash function.
- Classic space-time tradeoff:
  - If no space limits: trivial hash function with each key as an index;
  - If no time limits: trivial collision resolution with sequential search;
  - With space and time limits (the real world): design good hashing
- Disadvantages for hash tables?
  - Difficult to expand.
  - There is no order.
    - Hash table can be implemented by a balanced BST to keep keys in order. Operations will be in O(logn)
  - Memory efficiency?